# The Apex Developer Manual

“Any fool can write code that a computer can understand. Good programmers write code that humans can understand.” -Martin Fowler”

This document lists a working set of coding standards used as a reference when developing on the [Company] Salesforce org. Understanding and following these guidelines will be essential for submitting Pull Requests that get approved and merged, as they lay down the foundational review criteria.

The following descriptions are representative, not exhaustive, as many common sense variations can be derived. Understand the principals and apply them wisely.

## Development Process

We follow the development process below so that we can deliver value promptly for the business, and so that we can continue delivering increasing value for as long as needed.

The following assumes the simplest org without SFDX sophistication.

1. Begin with well-understood requirements, if not understood, ask questions and record understandings.
2. Develop in a development sandbox, with useful unit tests and developer testing.
3. Deploy Change Set to QA.
4. A different team members tests the code from the perspective of a System Admin, since that profile should be able to do everything.
5. Deploy Change Set to UAT.
6. The same tester or a different one tests the code from the perspective of impacted users, profiles, etc.
7. Deploy Change Set to Prod.

## Design Considerations

### Trigger Architecture

We follow a simple Trigger-Handler-Helper pattern. Triggers act as the first filter for DML context. Handlers check for conditional entry, collect records, and pass them off for handling. Helpers do the work and execute the logic.

Avoid design leakage, where helper work code creeps into the handler. This is also true for Triggers.

### Scheduled Jobs

We have the following scheduled jobs, or a link to an authoritative source. We use this to make decisions on future scheduled work.

### Integrations

We have the following integrations, or a link to a documentation repository. Some integrations are not as well-documented as the others. Creating documentation during real-time analysis is encouraged, just don’t get carried away. Usually, the notes and sketches we make to understand can be cleaned up and added to the documentation.

### Practiced Design Patterns

When business logic is shared, we create Service classes. We avoid getting carried away with abstraction and indirection, not being overly-ideological about DRY, SOLID, or Gang of Four. But when these are the right solution, we are happy to implement them. Conversations regarding design are encouraged.

### [Other Design Considerations]

## Naming Conventions

### Classes

Class names should define or describe a thing or entity, being simple and descriptive. A class name should be camel-cased. Acronyms and abbreviations should be used sparingly to avoid confusion. Do not use underscores or other special characters in a class name.

#### Class Variations

* Scheduled Apex Classes should be suffixed by ’\_Scheduled’;
* Batch Apex Class should be suffixed by ’\_Batch’.
* Controllers should be suffixed by ’\_Controller’.
* Test Classes should be suffixed by ’\_Test’.
* Interfaces should follow the same rules as Classes and be prefixed with a capital “I”.

### Variables

Variable names should be camel-cased and start with a lowercase letter first. This rule applies to instance and class variables. They should not start with an underscore or a dollar sign. Avoid using underscores or special characters in a variable name. Variables names should be meaningful and provide context/intent of their use. **Avoid** single letter variable names like (a, c, o). For loops with an index, the name can be shortened to i.

Here are the current exceptions to the above rule and are preferred abbreviations:

* pge = Priority Group Entry (Priority\_List\_Entry\_\_c)
* ocr = Opportunity Contact Role (OpportunityContactRole)
* pccMilestone = Pre-Closing Closing Milestone (Pre\_Closing\_Closing\_Milestone\_\_c)

### Constants

Constants, as a special type of variable, go by different rules. Constant name should be all uppercase, separating words by underscores. Avoid any special characters in a constant name. Use constants sparingly and with good purpose, as they can clutter a class up and be used with very little benefit. Things that don’t change often can be simple strings, with modern IDEs they are not hard to find or change.

### Methods

Method name should be a verb/action and provide context on what happens. Like variable names, they should be camel-cased with a lowercase first letter, in all situations.

## Code Layout and Formatting

### The Editor Config

Well formatted code increases readability, understanding and ultimately maintainability of the code base. A developer should strive to use a consistent layout and format. It will make the lives of other developers (who want to read, review, or maintain your code) a lot easier.

If you are using IntelliJ/IlluminatedCloud, an idea-editor-config.xml has been provided, which can easily be imported to conform to our general layout and formatting. If you are using VSCode, an .editorconfig file has been provided in the repo. To set it up, start with the following steps:

1. Pull from main and make sure .editorconfig is in the root directory.
2. Download the VSCode extension EditorConfig for VS Code.
3. Navigate to VSCode Settings->Extensions->EditorConfig->Template
4. Change from default to local the path for .editorconfig file.

### Line Wrapping

When an expression will not fit on a single line, break it according to these general principles:

Minor line overflows may be acceptable, as most screens are larger than the default line length. Use good judgement.

When a SOQL does not fit on one line well, break it apart by clause, and avoid putting all fields on different lines, as it makes the code harder to read without scrolling.

### Placement

In a method, place variable declarations close to usage, avoid jamming them all at the top of the method. This may be  
contrary to some, but I believe it makes the code easier to read, as variables can provide context. I also find it  
easier to reason about, as the usage/scope of a variable is clearer.

### Class and Method Declarations

When creating classes and methods, the following formatting rules should be followed:

* The opening brace { appears at the end of the same line as the declaration.
* The closing brace } is placed on a line by itself, indented to match its corresponding opening statement.
* There should not be a space between a method name and the parenthesis ( that starts its parameter list

### if, if-else, else-if, else Statements

If statements should always use braces {}. Avoid the following error-prone form, as it will be caught in PMD Scan:.

### Blank Lines

Blank lines improve readability by setting off sections of code that are logically related.

Use good judgement, prioritize readability, clarity, and context.

## Unit Tests

Unit Tests are important, so they get their own section. Well-formed, consistent, and rich unit tests are as valuable as the code itself. If they are conceived and created as an afterthought or annoyance that must be completed, it is likely that the intended value they can provide is greatly diminished.

### Naming Unit Tests

Test names should describe the test condition and outcome, to a reasonable extent. The name of the unit test should answer a question. If someone were to ask what was expected, the name provides the answer.

### The Structure of a Unit Test

Invariably, the test consists of the GIVEN, the WHEN, and the THEN.

* GIVEN - data setup
* WHEN - the test condition
* THEN - validate the outcome of the operation with an assertion

The statements in the WHEN (inside Test start and stop) should be minimal and focused on the operation. Do not allow setup code to creep in. Make sure you use Test.startTest() and Test.stopTest().

### Test Variable Names

It’s not often you end up in a Test Class unless you mean to. Therefore, we can dispense with the test prefix or postfix on record variables. Not testContact, but contact, or better primaryContact.

### Assertions

I wish I didn’t have to write this, but all Unit Tests must have assertions. Otherwise, what is the point of the unit test. This will be picked up by PMD.

Use the newer Assert class instead of the older System.assert()

There are times when a message is not needed as an assert parameter, and there are times when a message can add value. Typically, areEqual() and areNotEqual() do not need messages, while other methods do benefit from them. The message below serves little function.  
//totally unnecessary Assert.areEqual('Discovery', opportunity.StageName, 'The stage does not match Discovery');

## Opinionated Practices

These are key coding principles and practices that aim to produce efficient and scalable code.

### Keep code stupid simple

Sometimes, you’d think a developer was being paid on how many lines they wrote. Other times, you might think that the developer was barely confident their code would work at all.

Code should be simple, straight-forward, and so obvious that a comment is not required. But if a method does something non-obvious, then a comment should be added to provide context, background, implication, etc.

### On Maps

Maps can make difficult/convoluted code simple. Once you begin nesting loops, ask yourself if this could be accomplished with a map. The answer is usually yes.

### Use relationships to reduce queries

Take advantage of child sub-queries. They can save you from taking multiple trips to the database, as well as the code gymnastics that follows.

### The first pull request

Upon creation of your first pull request, prefix the PR name with brownM&M. Failure to do so can result in an immediate contract forfeiture with your company.

### Avoid Double Negatives

These are seen from time to time and they are totally pointless. They make the code harder to read and offer no value.

### Extract Logic with Methods

Methods should be small, atomic, and easy to understand. Often, these qualities are difficult to achieve, and the method grows in size. This will impact Complexity violations, so we need to have a way to reduce the complexity.

One useful approach is via private helper methods. Queries can take up a lot of space, extracting a method gives us a name that can describe the query logic and reduces the code in the primary method. This can be done with other forms of logic to good effect, like conditionals, providing context and easier readability/maintainability.

## Other Conventions

### Header Comments

Do not include headers. They offer little value, quickly go out of date, and the value they may provide is found by using version control.

### Various Declarations

* Use @IsTest instead of @isTest, same for other annotations
* Use SObject instead of sObject
* Match the casing on the api name Is\_Relevant\_\_c instead of is\_relevant\_\_c

## Logging

There is not currently a specific or mandatory requirement for logging. Follow these general guidelines.

* Do not leave System.debug() in your code. Use them for debugging, remove before committing.
* For code that would benefit from logging, use the installed and preferred Nebula Logger. If you have not come across this package before, their GitHub repo has excellent documentation and there are very good community created resources on the web.
* Use Logger.setScenario() to your benefit. It makes it easier to group like logs in the Log List View in the UI.